Highly Scalable Metadata Search and Indexing

Ethan L. Miller
Andrew Leung • Tim Bisson* • Minglong Shao* • Shankar Pasupathy*
Storage Systems Research Center
University of California, Santa Cruz
*NetApp

pdsi

Baskin Engineering
UC SANTA CRUZ

Challenge: scalable search mechanisms

- Indexing is a critical issue
  - Speed and effectiveness of search limit the usability of very large scale storage systems
- Very large scale indexes are often resource-intensive
  - Google and Yahoo have web-scale indexes, but they use thousands of processors to do it!
  - Performance is high (memory resident indexes)
- Indexing can take advantage of locality
  - Users typically aren't searching over the whole file system
  - Users may not have permissions to see everything
- Challenges
  - Building indexes that scale
  - Building less resource-intensive indexes
  - Building indexes that leverage locality
  - Incorporating security into indexes
  - Integrity: failure of a centralized index can be difficult to recover from

Challenge: gathering metadata for indexes

- Indexes are only as good as the information that goes into them
- Critical types of metadata include
  - Content
    - Domain-specific techniques for gathering it
    - May need domain-specific search mechanisms
  - Provenance
    - *How* was the data generated?
    - On what data and programs does this file depend?
- Challenges
  - How can provenance be tracked efficiently?
  - How can domain-specific metadata be handled?
    - Gathered?
    - Indexed as part of the file system?

Challenge: data mining in mass storage

- Large storage systems contain a lot of useful data
  - Can be difficult to fully utilize
- Traditional data mining techniques may not be effective
  - Infeasible to read out the entire storage system for data mining
- Two potential approaches
  - Index the data when it's written to storage, and use the indexes for mining
  - Distribute computation to the storage devices, allowing them to run in parallel

Our approach:partitioned indexes

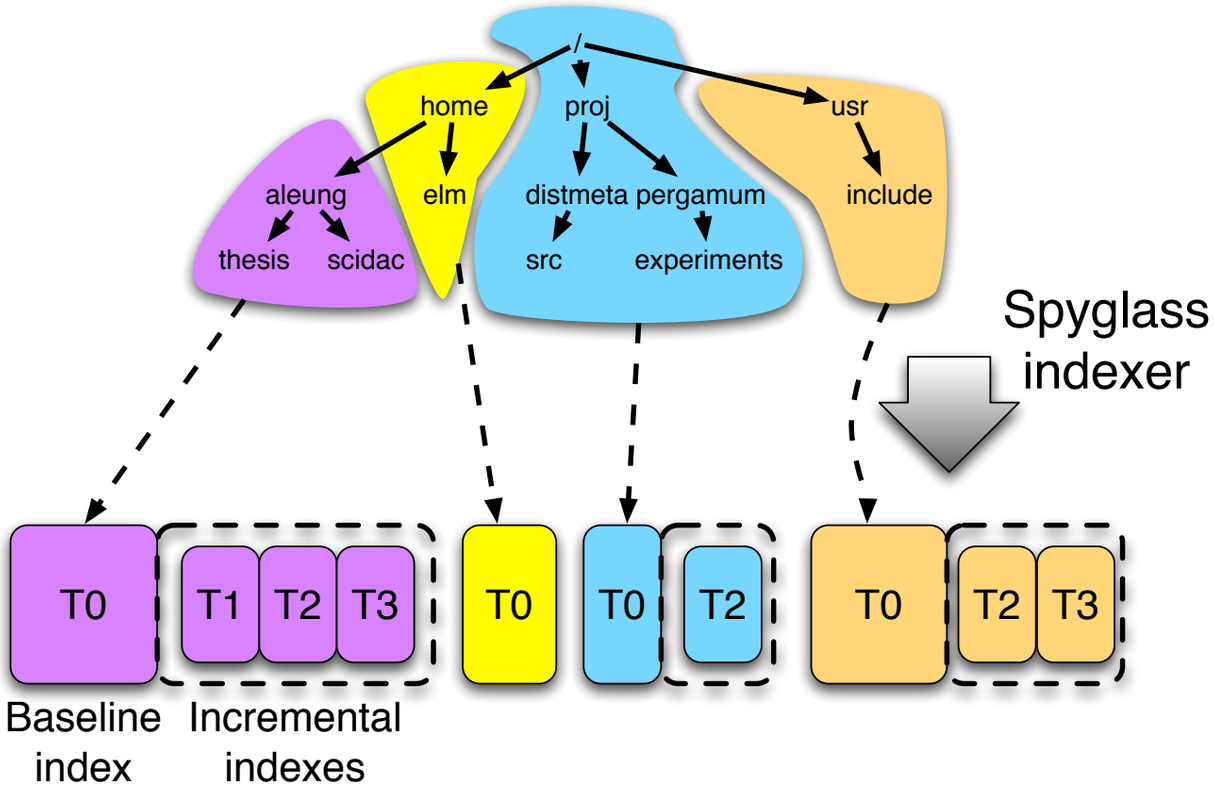
- Break up single large index into many smaller indexes
  - Each subindex covers a manageable amount of data
  - Individual subindexes can be searched quickly
  - Individual subindexes can be rebuilt after corruption (or to reflect incremental changes)

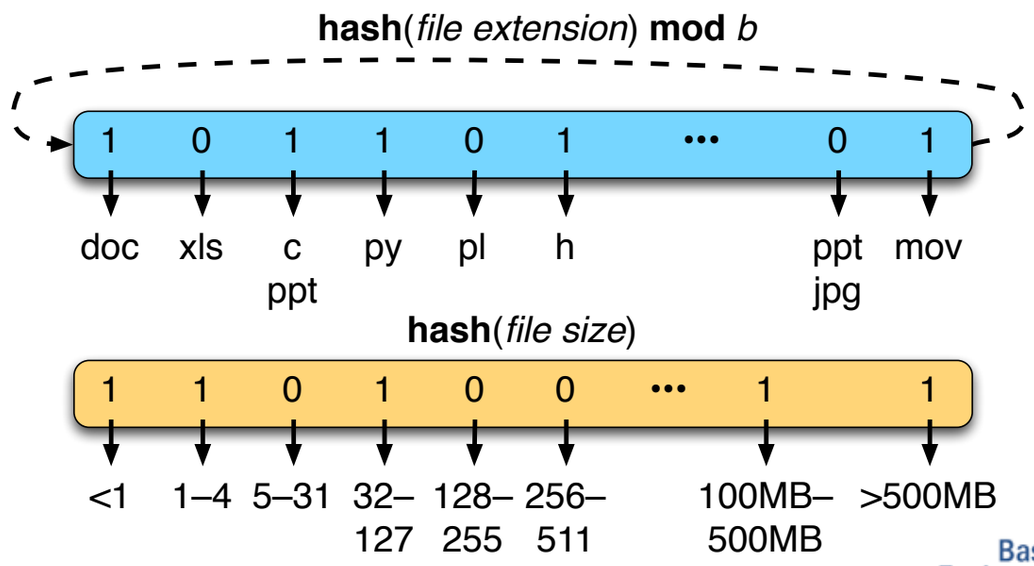
- Problem: can't search *all* subindexes for every query
  - No better than what we do today (and maybe worse)
- Problem: subindex search needs to be efficient
  - Can't cache *all* subindexes in memory

- Leverage locality?

Spyglass design

- Partition file system hierarchy by subtree
  - Each subtree is an independent subindex
- Summarize contents of each subindex
  - Quickly rule out entire subindexes that can't satisfy the query
- Log incremental changes
  - Rebuild index when there are "enough" changes
- Integrity is much easier
  - Rebuild subindex, not entire index





Current status

- Prototype implemented for attributes
  - File size, type, owner, etc.
  - Content being done now...
- Used k-d trees for individual indexes
  - Straightforward to optimize tree structure for each individual subtree
  - Recently-accessed subtrees cached in memory
    - Only takes a few milliseconds to read and search a tree
- Implementation tested using 300 million files crawled at a "large storage company"
  - Locality really helps!
- Performance is very good
  - Compared against standard databases
  - Queries on Spyglass were 10–6000 times faster!

Ongoing research

- Extending attribute-based search to content
  - Will locality help as much?
  - Will compression help to reduce the size of the indexes?
- Scaling indexes to trillions of files
  - Existing systems (databases or otherwise) can't handle such large systems without massive amounts of hardware
  - Will summarization work with millions of indexes?
- Non-hierarchical file systems
  - Use similarity (or something else) to group files?
    - Talk on this earlier today
- Extending indexing to archives